Санкт-Петербургский государственный университет

**Кафедра информационных систем**

**Кисляков Владислав Сергеевич**

**Выпускная квалификационная работа бакалавра**

# Информационная система идентификации материала по данным его спектра

Направление 010400

Прикладная математика и информатика

Научный руководитель,
доктор физ.-мат. наук,
профессор
Матросов Александр Васильевич

Санкт-Петербург
2017

# Содержание

# Введение

В данной работе решается задача идентификации материалов по данным их спектров. В качестве исследуемых образцов выбрана древесина, так как она является сложным по составу материалом и, в случае успешного результата, представленный в данной работе алгоритм можно применять к другим материалам со сложным составом. Данные получены для трех различных типов светового излучения.

Для решения этой задачи была использована композиция классификаторов на основе нейронных сетей, устроенных по типу многослойного персептрона (multilayer perceptron) [3], для предварительной подготовки данных применялось сингулярное разложение (Singular Value Decomposition, SVD) [4][5] и метод главных компонент (Principal Component Analysis)[6].

Ранее применяемые методы спектрального анализа основывались на сравнении. Полученные данные соотносились с эталонами из базы и определялся наиболее близкий по спектру класс [2]. Так же в работе [1] был разработан метод, учитывающий статистическую информацию об отдельных породах, на основе чего строит, так называемые, «коридоры» с верхними и нижними значениями границы расположения спектра. Так, для всех спектров образцов одной породы строится «коридор», в границах которого они находятся. Далее алгоритм для каждого нового образца ищет наиболее вероятное местоположение из всех таких коридоров [1]. Однако, зачастую спектры материалов одного и того же типа могут расходиться более, чем от спектров других типов. То есть «коридор» в рамках одного типа не всегда можно построить, так же они могут содержаться один в другом.

Так как необходимо сравнивать данные со всеми образцами из базы, данные методы работают очень долго и не всегда удается получить качественный результат на новых примерах.

Одной из проблем распознавания материала по спектрам является некачественность получаемых данных в процессе их сбора [2]:

- помехи и шумы, связанные с погрешностями аппаратуры;
- влияние таких факторов как температура, освещённость и т.п.;
- неоднородность образцов одного типа в виду различных причин.

Преимуществами спектрального анализа являются [2]:

- анализ небольшого количества материала, без его разрушения;
- универсальность методов анализа;
- большая точность и производительность.

# Постановка задачи

Решаемая в данной работе задача заключается в разработке алгоритмических средств и программного обеспечения для идентификации материала по его различным типам спектров. Так же требовалось учесть возможность как добавления новых типов спектра, так и сужения их количества для различных материалов. Для этого необходимо решить следующие задачи:

1. провести предобработку данных для устранения шумов и снижения размерности;
2. реализовать нейронные сеть по типу многослойный персептрон;
3. на основе нейронных сетей построить композицию классификаторов для идентификации классифицируемых спектров.
4. выполнить тестирование на основе спектров

В качестве тестируемого материала использовались данные трех различных типов спектра древесины, полученные для инфракрасного, видимого и светодиодного излучений.

# Обзор литературы

Базируясь на материал изложенный в [2] [11] удалось составить первое впечатление об исследованиях, касающихся спектрального анализа. Основные исследования в области анализа спектров касались в основном качественного анализа, то есть выявлением того, какие элементы входят в состав анализируемого образца. Распознавание же материала в целом основывалось на качественном анализе с применением эталонов, т.е. классификация основывалась на том, какие элементы входят в состав вещества.

Авторами работы [1] был представлен метод, не основывающийся на анализе состава. Перед ними стояла задача распознавания древесины. В виду особенностей исследуемого ими материала, стандартные методы не принесли бы качественного результата, так как породы древесины практически идентичны по своему составу. Метод представленный в [1] основывался на статистическом анализе спектров отдельных пород и составлял «коридоры» с верхней и нижней границами спектра. Однако авторами не был учтено, что границы одной породы могут поглотится границами другой.

В последнее время стало популярным использование нейронных сетей в виду их невероятной способности к обобщению данных. Получить начальное представление об устройстве нейронных сетей удалось в [7]. Более же подробное описание представлено в [3], из которой были подчерпнуты методы работы многослойного персептрона и алгоритм его обучения. Описание обобщающих свойств нейронных сетей в [3] и [7] дало понять, что они способны избавиться от недостатков, представленных выше методов.

Однако возникла проблема сжатия данных для подачи нейронной сети. Решение было найдено в книге [8] в описании работы метода главных компонент. Более подробно разобраться в алгоритме сжатия методом главных компонент позволила работа [6], так же из нее удалось узнать, как свести его к нахождению сингулярных значений матрицы, подробное описание которого представлено в [5].

Идея раздельного обучения сетей для каждого типа излучения и дальнейшего объединения их в единую систему распознавания путем композиции, была частично подчерпнута их материалов, представленных в [13].

# Глава 1. Алгоритм решения задачи

## 1.1 Описание алгоритма

Основной задачей является обучение нейронной сети, но прежде требуется подготовить данные. Во-первых, необходимо сократить размерность данных и нормализировать их, так как в виду особенностей устройства нейронных сетей, вектора большого размера и со значениями далеко не близкими к нулю потребуют значительного времени для качественного обучения сети и не позволят достичь желаемой точности [8]. Во-вторых, требуется устранить шумы, возникшие в виду несовершенства методов получения спектра с материала. Обе эти задачи решает метод главных компонент сведенный к сингулярному разложению матрицы данных. Таким образом, разделив данные на обучающую, тестовую и контрольную выборки, достаточно провести сингулярное разложение для обучающей выборки, а для остальных выборок получить необходимый результат простым перемножением матриц.

Подготовив данные перейдем построению классификатора. Поскольку данные были получены для трех типов светового излучения, обучим для каждого из них отдельную «спектральную» сеть (обученная на отдельном типе спектра) и на основе них составим комбинацию классификаторов для получения наилучшего результата за счет компенсирования недостатков отдельных типов спектров остальными. Обучив сети составляется обучающее множество из ответов трех сетей, путем их соединения (14 типов древесины, тогда получается вектор размерностью 42). На основе нового множества обучается конечная сеть. При тестировании на контрольном множестве «спектральные» сети ставятся перед конечной и передают ей объединенный вектор своих выходных сигналов. Количество «спектральных» сетей может быть расширенно в случае появления данных с новыми типами спектра.

Обучение каждой «спектральной» сети заключается в нахождение наиболее информативного количества компонент пред обработанных данных

и количества нейронов на скрытом слое обеспечивающих достаточное обобщение.

## 1.2 Многослойный персептрон

Единицей обработки и информации в нейронной сети является нейрон. На блок-схеме рис.1 представлена модель нейрона. В этой модели можно выделить три основных элемента.

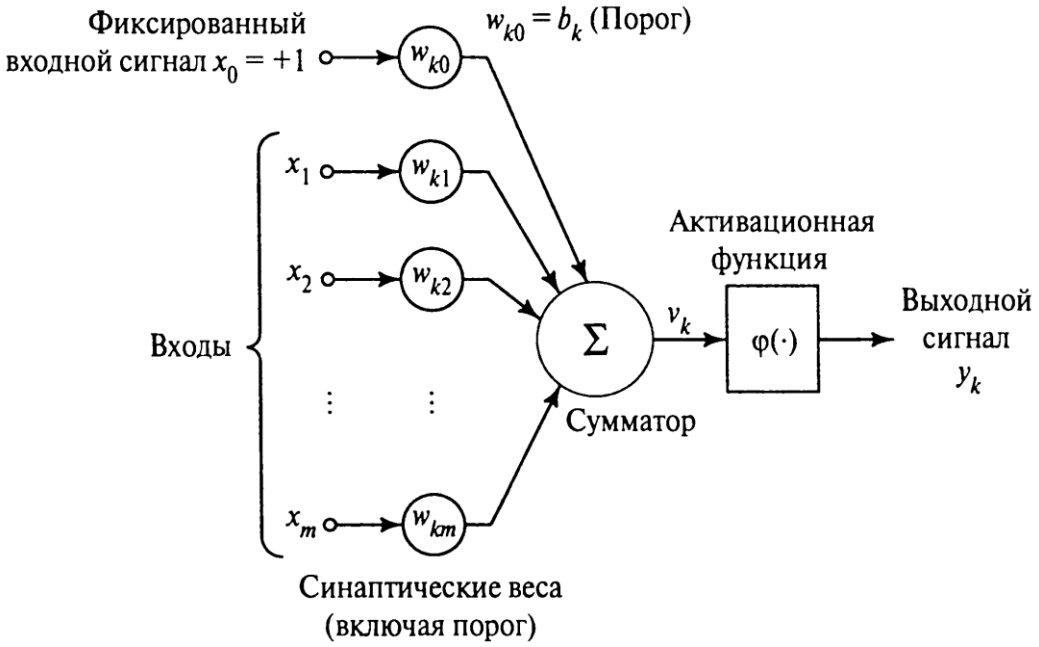



Рис. 1. Модель нейрона

1. Набор синоптических связей, каждая из которых характеризуется своим весом. Сигнал $x_j$ на входе синапса $j$, связанного с нейроном $k$, умножается на вес $w_{kj.}$
2. Сумматор складывает входные сигналы, перемноженные с соответствующими весами.
3. Функция активации ограничивает амплитуду выходного сигнала. Обычно нормализованный диапазон амплитуд выхода нейрона лежит в интервале [0, 1] или [-1, 1].

В модели нейрона на рис.1 включен пороговый элемент, обозначенный символом $b_k$. Он отражает увеличение или уменьшение входного сигнала, передаваемого на функцию активации.

Математически функционирование нейрона $k$ можно описать парой уравнений:

$$v_k = \sum_{j=0}^{m} w_{kj} \cdot x_j,$$

$$y_k = \varphi(v_k),$$

где $x_1, x_2, \ldots, x_m$ – входные сигналы; $w_{k1}, w_{k2}, \ldots, w_{km}$ – синаптические веса нейрона $k$; $v_k$ – функция активации; $y_k$ – выходной сигнал нейрона[3].

В качестве функции активации в данной работе используется сигмоидальная униполярная функция, определяемая логистической функцией

$$y_k = \varphi(v_k) = \frac{1}{1+exp(-v_k)}.$$

Так же в последствии нам понадобится производная активационной функции:

$$\varphi'(v_k) = \varphi(v_k) * (1 - \varphi(v_k))$$

Сигмоидальный нейрон – это нейрон, использующий в качестве функций активации сигмоидальные униполярные и биполярные функции.

Многослойный персептрон – это многослойная сеть прямого распространения, состоящая из сигмоидальных нейронов. Нейроны одного слоя не связаны друг с другом, каждый нейрон имеет связи со всеми

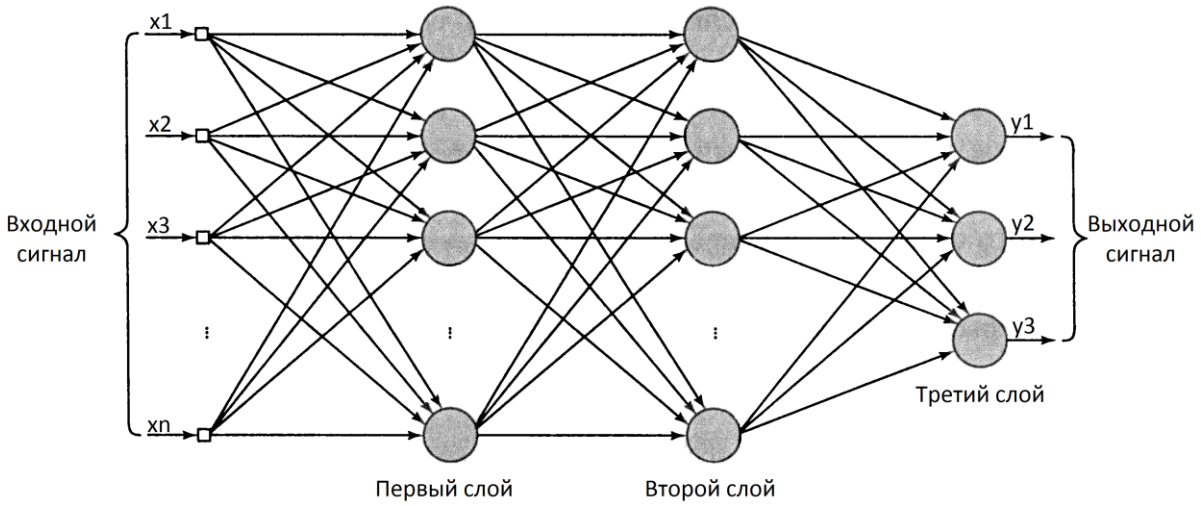



Рис. 2 Многослойный персептрон

нейронами предыдущего слоя и следующего. Сигнал передается по сети только в прямом направлении, слева направо, от слоя к слою. На рис.2

изображен архитектурный граф многослойного персептрона с двумя скрытыми слоями и одним выходным слоем [3].

Выходные нейроны составляют выходной слой сети. Остальные нейроны относятся к скрытым слоям. Таким образом, скрытые узлы не являются частью входа или выхода сети – отсюда они и получили свое название. Первый скрытый слой получает входной сигнал. Результирующий сигнал первого скрытого слоя, в свою очередь, поступает на следующий скрытый слой, и т.д., до самого конца сети [3].

Дальнейшие понятия, связанные с многослойным персептроном, такие как постановка задачи обучения и функция ошибки, будут приведены в следующем параграфе.

## 1.3 Алгоритм обратного распространения ошибки.

Алгоритм обратного распространяя слишком громоздок, потому будут приведены лишь конечные формулы без их выведения.

Поставим задачу обучения в интерполяционной формулировке:

Дано:

- Обучающее множеств: M = $\{(x^i, d^i)$, $x^i = (x_1^i, \dots, x_n^i)$, $d^i = (d_1^i, \dots, d_m^i)\}$, где $x^i$ - входной сигнал, $d^i$ - правильный ответ;
- Число $\varepsilon > 0$

Найти: веса, так чтобы $\forall i = 1, k \left\| F(x^i) - d^i \right\| \leq \varepsilon$, где F – ответ сети.

Введем понятие «ошибки»:

$$e_i(w) = \frac{1}{2}\sum_{j=1}^{p}\left(y_j - d_j^i\right)^2, \quad (1)$$

$$E(w) = \sum_{j=1}^{k} e_i(w), \quad (2)$$

где (1) - ошибка на одном примере, (2) - ошибка на всем множестве.

Введем понятие последовательного режима обучения. При последовательном режиме на каждом шаге на вход сети подается один

произвольно выбранный пример и веса изменяются в направлении целевой функции (1). После того, как каждый пример будет подан хотя бы один раз заканчивается эпоха и вычисляется (2) [3]

Веса изменяются по формуле:

$$w(t+1) = w(t) - \alpha(t)\nabla E$$

где t – номер шага цикла, $\alpha(t)$ – шаг обучения, $-\nabla E$ – антиградиент.

Для нахождения шага обучения используется метод наискорейшего спуска и $\alpha(t)$ определяется из аргумента:

$$\alpha(t) = \underset{\alpha}{arg\min} E(w(t) - \alpha\nabla E),$$

для решения этой задачи используется метод золотого сечения.

Антиградиент $\nabla E = (\frac{\partial E}{\partial w_{1\,1}^{1}}, \dots, \frac{\partial E}{\partial w_{i\,j}^{l}}, \dots)$ вычисляется методом обратного распространения ошибки. Элементы вектора градиента вычисляются по формуле:

$$\frac{\partial E}{\partial w_{i\,j}^{l}} = \delta_j^l y_i^{l-1},$$

где $l$ - номер слоя; $j$ - номер нейрона на слое $l$; $i$ – номер элемента выходного сигнала предыдущего слоя; $y_i^{l-1}$ – эл-т выходного сигнала предыдущего слоя (для первого слоя берется входной сигнал сети); $\delta_j^l$ – локальный градиент. Нам известны $y$ на каждом слое, включая входной сигнал, однако локальный градиент требуется вычислить. Для последнего слоя локальный градиент вычисляется в первую очередь и вычисляется справа налево, слой за слоем, используя полученные значения со слоя справа. Локальный градиент последнего слоя находится по формуле:

$$\delta_j^l = (y_j^l - d_j^i)\varphi'(v_j^l).$$

Формула для остальных слоев:

$$\delta_j^l = \left(\sum_{k=1}^{P_{l+1}} \delta_k^{l+1} w_{j\,k}^{l+1}\right)\varphi'(v_j^l),$$

где $P_{l+1}$ – кол-во нейронов на следующем слое [3].

## 1.4 Метод главных компонент и сингулярное разложение

Метод главных компонент - наиболее распространенный подход к снижению размерности данных, с потерей наименьшего количества информации. Вычисление главных компонент можно свести к нахождению сингулярного разложения матрицы данных.

Суть сингулярного разложения заключатся в привидении матрицы $A$ размера $m \times n$ к виду $A=U*S*V^T$, где $U$ и $V$ - унитарные матрицы размера $m \times m$ и $n \times n$ соответственно, $S$ - диагональная матрица с вещественными положительными числами на диагонали. Диагональные элементы матрицы $S$ называются сингулярными числами матрицы $A$, а столбцы матриц $U$ и $V$ левыми и правыми сингулярными векторами соответственно [4][5].

Переход к методу главных компонент заключается в получении матриц $T = U*S$ и $P=V$, тогда $A = T*P$. Матрица T - это представление исходных данных в пространстве главных компонент (строки - объекты, столбцы - новые признаки, вместо точек спектра). Оставляют в матрице T лишь первые $k$-столбцов (главных компонент), остальные удаляют. Аналогично - в матрице $P$. Чтобы перейти к пространству главных компонент для новых данных необходимо $T\_test = A\_test * P$, где $A\_test$ – новые данные; $T\_test$ – представление $A\_test$ в пространстве главных компонент [6].

## 1.5 Композиция классификаторов

В оригинальном смысле композиция классификаторов подразумевает под собой применение различных по типу классификаторов для компенсации недостатков отдельного метода другими. Применяема же в данной работе композиция скорее более близка к методам, применяемым в распознавании изображений, когда проводиться обучение сетей для распознавания отдельных элементов изображения и в последствии они выстраиваются перед сетью, классифицирующей изображение полностью. Или же можно назвать

данную архитектуру неполносвязной нейронной сетью с пред обучением вешних слоев [13].

Основная идея заключается в том, чтобы получить от каждого типа спектра наилучший результат классификации, который он способен предоставить. Так же в случае неправильного ответа от каждого классификатора имеется возможность на выходе конечного классификатора получить верный ответ.

# Глава 2. Описание программного обеспечения

В рамках данной работы автором было реализовано программное обеспечение, позволяющее решить поставленные задачи, методами, описанными в главе 1.

## 2.1 Описание архитектуры программного обеспечения

Данный программный продукт написан на компилируемом строго типизированном языке программирования общего назначения C++ с использованием фреймворка Eigen [4]. В качестве среды разработки использовалась Visual Studio [10].

Eigen - библиотека линейной алгебры для C++ с открытым исходным кодом. Написана на шаблонах и предназначена для векторно-матричных вычислений и связанных с ними операций. Она содержит класс SVD производящий сингулярное разложение матрицы. В работе был использован метод JacobiSVD, использующий метод Якоби для нахождения собственных значений матрицы [4].

Для решения поставленных в работе задач необходимо было построить программных продукт таким образом, чтобы архитектура позволяла относительно легкую масштабируемость, изменение деталей обучения и количества исследуемых типов спектра. Для этого посредствам классов были написать следующие модули:

1. Модуль предобработки данных, обеспечивающий реализацию алгоритма, описанного в параграфе 1.4. Реализован в классе mobuleSVD.
2. Модуль считывания данных, считывает, реализует разбиение и кросс валидацию множества. Реализован в классе DataSet.
3. Модуль нейронной сети, позволяющий обучать сеть по входным данным, отвечать на входной сигнал и настраивать параметры обучения. Реализован в классе Net.

4. Модуль спектрального классификатора, агрегирующий выше указанные модули и реализующий алгоритм из параграфа 1.1, т.е. содержит в себе сети, как вспомогательные, так и основную. Так же модуль должен позволять расширение кол-ва «спектральных» сетей.

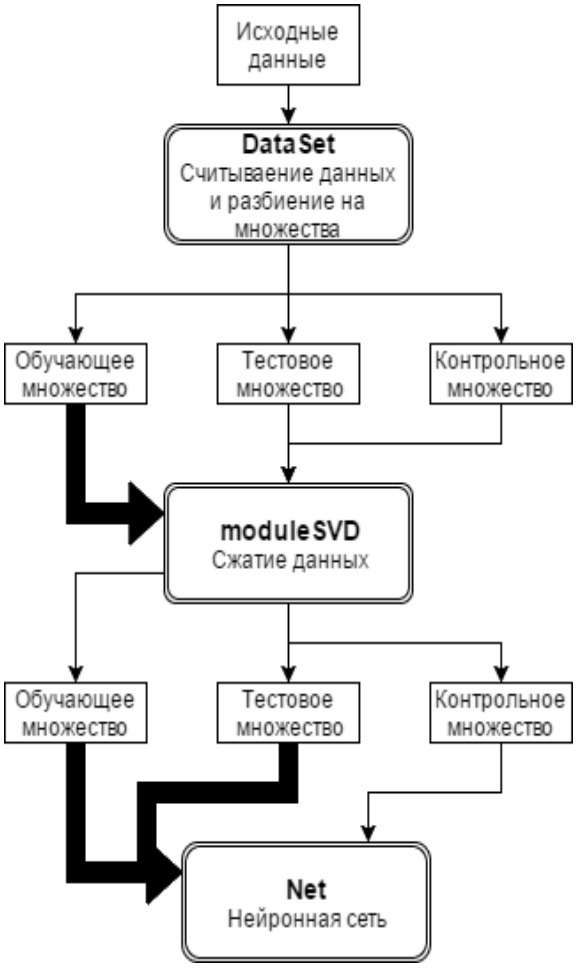



Рис. 3 Блок-схема взаимодействия модулей

Реализован в классе SpectralClassifier

Класс DataSet описывает данные, производит их сбор и разбиение на множества: обучающее, тестовое и контрольное. Класс mobuleSVD производит сжатие данных используя Eigen\SVD и алгоритм из параграфа 1.4 [4]. При создании объекта класса mobuleSVD требуется отправить обучающую выборку и в конструкторе класса, тогда будут созданы преобразующие матрицы, затем отправив требующую сжатия выборку с помощью метода getNewhData указав количество компонент можно получить сжатые данные и обучающую методом getTeachData. Класс Net описывает многослойный персептрон (указав в конструкторе класса один слой можно получить

однослойный персептрон) и методы для работы (обучения и получения ответа от обученной сети) с ним, также он агрегирует класс Layer, а он в свою очередь Neuron, описывающие слой сети и искусственный нейрон соответственно. Класс Net позволяет корректировать настройку параметров остановки обучения сети: необходимую величину ошибки на обучающем множестве, расхождение ошибки обучающего и тестового множества, максимальное количество эпох. Блок схема на рис. 3 схематически отражает порядок взаимодействия модулей. Толстыми линиями показаны данные, которые поступают в модуль в первую очередь один раз для его активации к работе, а тонкими данные которые подаются в последствии для получения результата работы модуля [9]. Каждый из модулей 1.1-1.3 независим и может быть применен для выполнения своей задачи в других работах.

Класс SpectralClassifier является ключевым, он реализует основной алгоритм данного исследования (параграф 1.1). Классу необходимо задать параметры сетей (кол-во нейронов и главных компонент), которые он обучит по схеме рис. 3 и образует для себя обучающее множество. Затем после вызова метода teachSR обучится сам агрегируя в себе класс Net [9].

.

# Глава 3 Обучение сети и подбор компонентов

## 3.1 Выбор условий остановки обучения

Важным элементом обучения нейронной сети является условие остановки обучения. Как привило основным критерием является достижение необходимой погрешности на обучающем множестве. Однако это не всегда я является показателем качества обучения, так как возможны случаи переобучения сети, т.е. возрастание погрешности тестового множества, или же остановка снижения погрешности тестового множества при одновременном уменьшении погрешности обучающегося.

Для выявления необходимых условий остановки были проведены предварительные тесты в ходе которых удалось определить их.

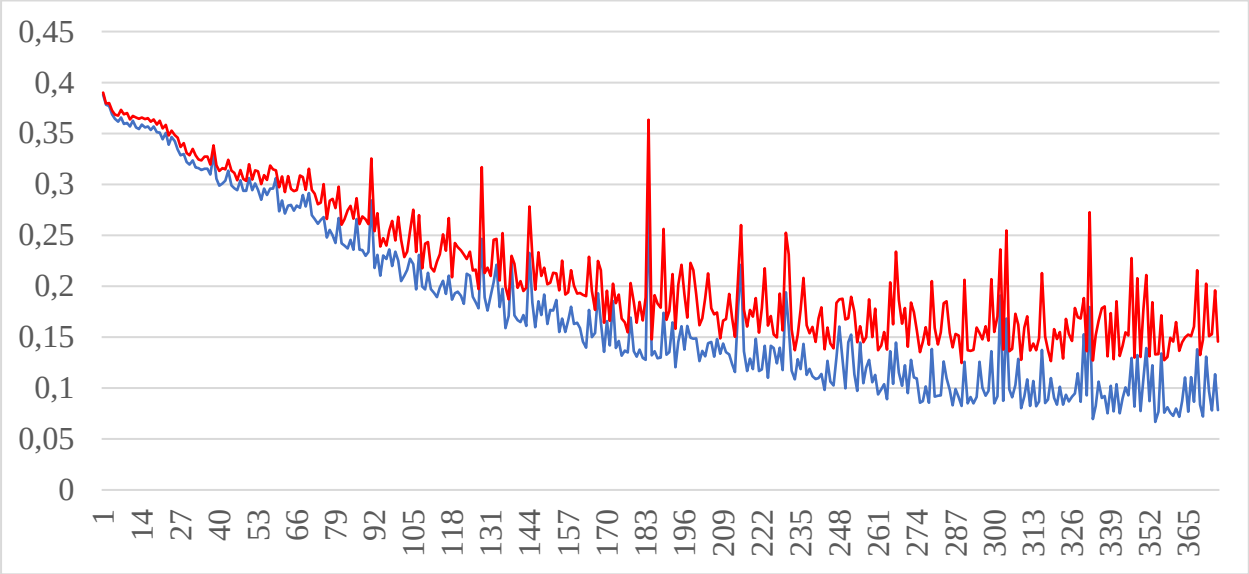



График 1. Изменения погрешности сети, обученной на данных в инфракрасном диапазоне спектра

Поведение графиков изменения погрешностей сетей обучаемых для отдельных диапазонов практически совпадает, потому рассмотрим график 1 как образец, на котором красная кривая - отражает погрешность тестового множества, а синяя – погрешность на обучающем. Проводилось обучение сетей с различным количеством компонент и нейронов на скрытом слое для выявления закономерностей поведения. Было отмечено, что погрешность на обучающем множестве почти везде достигала определенного значения и

продолжала снижаться гораздо медленнее, можно сказать практически прекращала снижаться. Так же было отмечено, что погрешность тестовой выборки с некоторого момента начинала расходиться с погрешностью обучающей, однако возрастания не наблюдалось, она продолжала снижаться, но с гораздо меньшей скоростью, чем в начале. Данное положение можно наблюдать на графике 1.

Проанализировав значения погрешностей на всех сетях были предложены критерии остановки. Первыми условиями можно отметить достижение необходимой погрешности на обучающем множестве для каждой сети:

- видимое излучение: 0.06
- инфракрасный диапазон: 0.07;
- светодиодное излучение: 0.19.

В данные значения не будут изменены в виду того поиск количества компонент и нейронов на скрытом слое будет нацелен на снижение разницы между погрешностями. Так же чтобы увеличит скорость проведения кросс валидации введем ещё одно условие остановки. Обучение будет останавливаться если разница между погрешностями сильно возрастет. В конце каждой эпохи будет вычисляется среднее значение разности погрешностей на протяжении нескольких эпох и в случае превышения этого значения 0.08 обучение будет останавливаться. Данная мера позволит долго не проводить обучение не способных качественно снизить погрешность тестовой выборки и выделит на фоне них сети с наименьшей разницей погрешностей. Задействовать условие, основанное на проверке остановке изменения величины погрешности, не представляется возможным, так как можно заметить, на графике 1 видно, что погрешности «скачут» и вычисление среднеквадратического отклонения не принесет желаемого результата.

Подобные тесты так же были проведены для конечной сети. Однако результаты немного отличались от полученных на вспомогательных. Так, например, на графике 2 можно наблюдать переобучение сети и отсутствие

сильных скачков погрешности. График 2 вполне отражает общее поведение погрешностей для всех предварительных тестов.

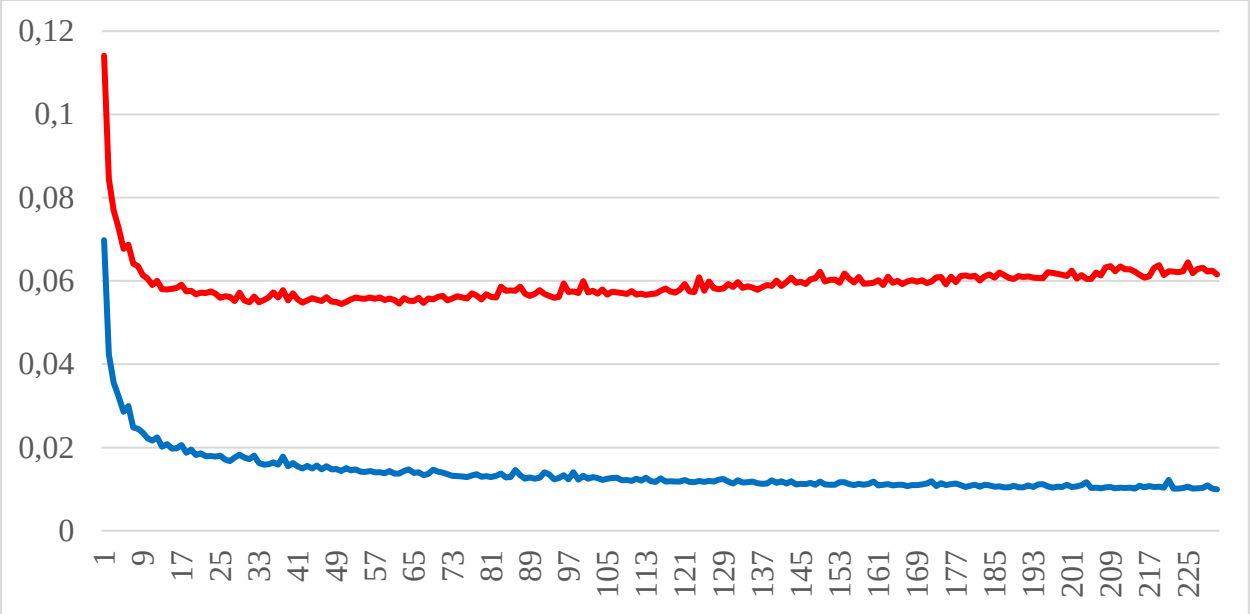



График 2. Изменения погрешности основной сети

Проанализировав значения погрешностей на проведенных тестах была предложена величина необходимой погрешности обучающей выборки равная 0.015. Так же для данной сети применен критерий остановки при отсутствии изменения погрешности, т.е. вычисляется среднеквадратическое отклонение погрешности на нескольких эпохах и в случае его малого значения обучение останавливается.

## 3.2 Выбор главных компонент

Необходимость выбора наиболее информативного количества компонент обусловлена тем, что, после проведения сингулярного разложения и приведения к пространству главных компонент, данные представлены в форме «сигнал/шум», т.е. до определенной компоненты находится качественный сигнал, а далее идет шум. Существует несколько эвристических алгоритмов для выбора количества компонент, но нам требуется более точный результат. Потому их количество для каждой вспомогательной сети определено экспериментально.

Суть эксперимента заключается в том, чтобы для различного количества компонент обучить сети с применением кросс валидации для получения

среднего результирующего для различных выборок. Для выбора количества нейронов воспользуемся эвристическим правилом геометрической пирамиды, по которому их количество в двухслойном персептроне определятся по формуле:

$$k = \sqrt{nm},$$

где $k$ – искомое количество нейронов; n – число входных данных; m – количество нейронов на последнем слое [15].

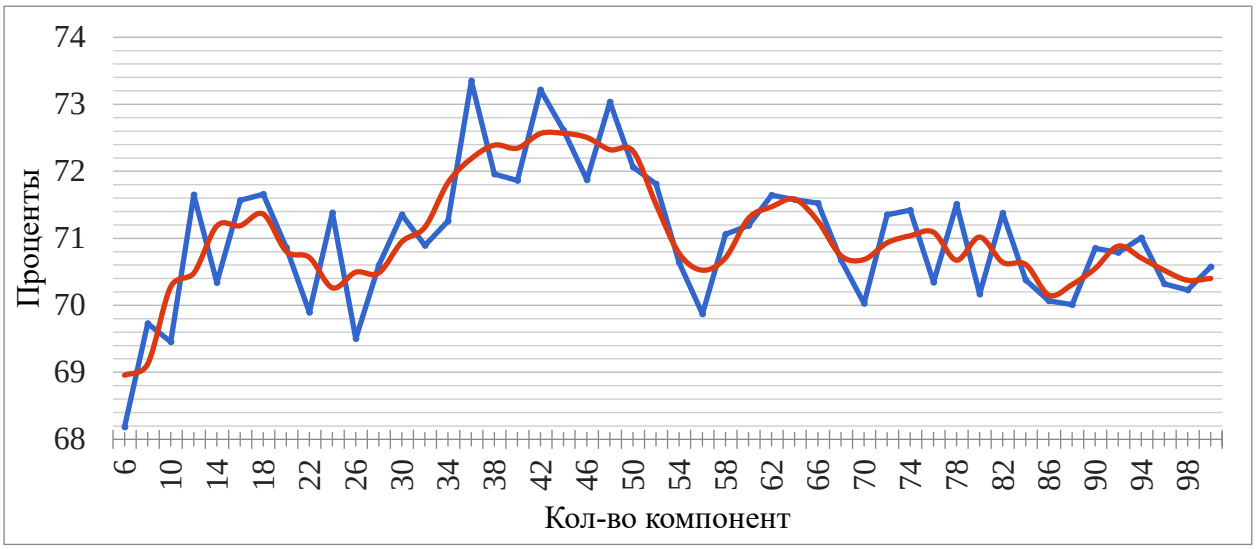



График 3. Зависимость точности на тестовом множестве от кол-ва компонент для сети, обученной на данных в видимом диапазоне спектра

На графике 3, где синяя кривая - отражает полученные значения точности на тестовом множестве в процентах; красная кривая – сглаживает значения; явно выделяются пиковый диапазон значений. Для остальных сетей прослеживается та же тенденция образования пиковых диапазонов.

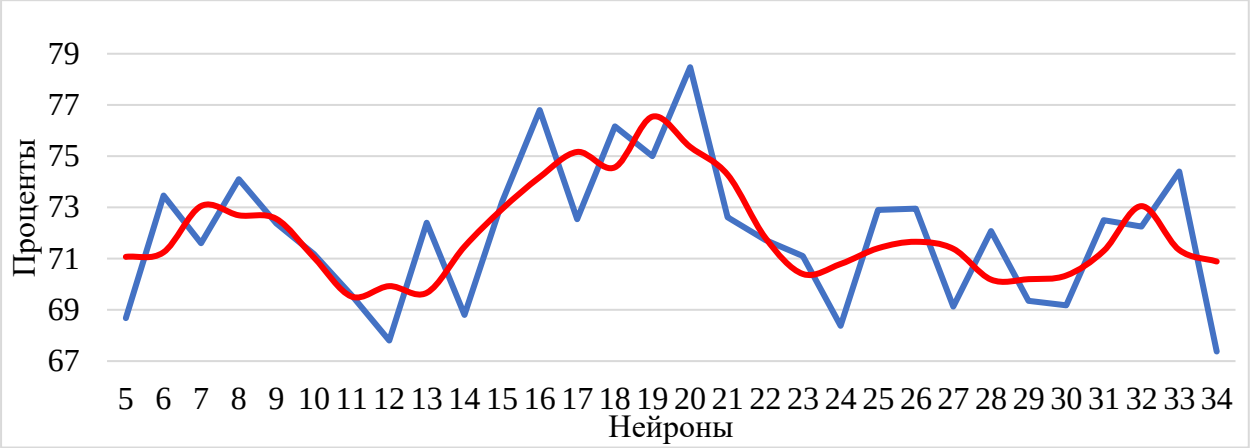
В качестве количества компонент выбраны центральные точки пиковых диапазонов:

- видимое излучение: 42
- инфракрасное излучение: 24;
- светодиодное излучение: 24.

## 3.3 Подбор количества нейронов

Выбрать наилучшее количество нейронов, так же является важной задачей. Поиск производится по той же схеме, что и в предыдущем параграфе.

Только теперь количество компонент известно, число нейронов будет изменятся. Дабы ускорить тестирование и более явно выделить пик в условии остановки обучения, основаном на разности погрешностей, значение приемлемой разницы было снижено до 0.06, что привело к уменьшению общей




обучаемости, но выделило облаости повышения точности.

График 4. Зависимость точности на тестовом множестве от кол-ва нейронов для сети, обученной на данных в инфракрасном диапазоне спектра

На графике 4, где синяя кривая - отражает полученные значения точности на тестовом множестве в процентах; явно выделяется пиковый диапазон значений. Остаётся лишь выбрать центральное значение пикового диапазона. Для остальных сетей прослеживается та же тенденция образования пиковых диапазонов.

В качестве количества нейронов выбраны центральные точки пиковых диапазонов:

- видимое излучение: 24
- инфракрасное излучение: 19;
- светодиодное излучение: 15.

# Глава 4 Тестирование программы

## 4.1 Результаты, полученные на контрольном множестве

Найденные количество компонент и нейронов должны обеспечить высокий проценты правильных ответов для сетей, распознающих материалы на отдельных спектрах. Конечная же сеть должна улучить эти результаты. В качестве финального классификатора использовался однослойный персептрон.

| Номер разбиения | Инфракрасное излучение | Светодиодное излучение | Видимое излучение | Финальная сеть (однослойная) |
|---|---|---|---|---|
| 1 | 81,9 | 54,16 | 71 | 91,66 |
| 2 | 73,6 | 49,3 | 61 | 90,2778 |
| 3 | 84 | 54 | 76 | 94,32 |
| 4 | 81,9 | 53,4 | 75 | 91,6667 |
| 5 | 83,9 | 53,14 | 79 | 94,32 |
| 6 | 85 | 58,7 | 79 | 94,32 |
| 7 | 75 | 51 | 67 | 96,52 |
| Среднее значение | 80,75 | 53,38 | 72,57 | 93,29 |

Таблица 1. Результаты, полученные на контрольном мн-ве

Таблица 1 отражает результаты обучения сетей с выбранными ранее параметрами и основной сети. Значения точности для вспомогательных сетей показывают не плохие результаты с учетом того, что древесина весьма сложный материал для анализа, так как многие породы очень близки по своему составу. Однако конечный классификатор показывает прекрасные результаты, повышая точность минимум на 10%, также на отдельных разбиениях выборки более 20%. В среднем же повышение точности приблизительно равно 13%.

Так же было проведено тестирование на многослойных сетях (двухслойный персептрон), с целью улучшить средний результат основной сети.

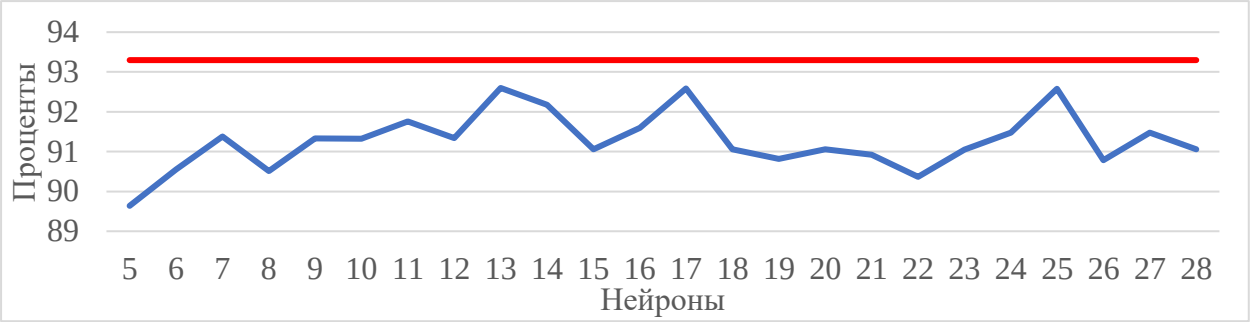



График 5. Сравнение результатов многослойной сети с различным кол-м нейронов и однослойной сети.

На графике 5 представлены результаты тестирования многослойной сети с различным количеством нейронов и однослойной сети, обе выполняли задачу объединения результатов вспомогательных сетей. Синяя кривая - отражает результат многослойных сетей; красная линия – среднее значение результатов однослойной сети. Как видно однослойная сеть лучше справилась со своей задачей. На отдельных выборках многослойные сети показывали результат, превышающий значение однослойной, но лишь на пол процента и таких случаев было мало, и они не были сосредоточены на сети с определенным количеством нейронов.

## 4.2 Вывод

Таким образом, анализируя каждый спектр в отдельности, мы повышаем его информативность, тогда конечный результат позволяет оценить исследуемые материалы в целом гораздо лучше, нежели, объединяя данные спектров в единое множество для анализа.

Опираясь на доказательства, приведенные в [13] можно сделать вывод, что повышение количества типов спектров может привести к увеличению процента правильных ответов метода. В данном случает отметим, что реализованное программное обеспечение позволяет добавить любое количество новых вспомогательных сетей, анализирующих типы спектров, что позволит улучшить производительность конечной сети [13].

.

# Заключение

В результате данной работы удалось создать новый эффективный метод распознавания материала по данным его спектра. Его успешное тестирование на спектрах древесины дает уверенность, что распространение данного алгоритма на другие материалы так же приведет к хорошим результатам. Основой данного вывода является сложность древесины как материала для спектрального анализа, в виду того что большинство пород древесины похожи.

Написано программное обеспечение, позволяющее распознавать материалы. Программный продукт может послужить основой для соответствующего оборудования. Так же продукт спроектирован таким образом, что каждая его часть независима, и, минимальными правками, любой модуль можно заметить в случае обнаружения более эффективного.

Данный алгоритм можно дополнить другими типами спектров, что теоретически позволит увеличить его результативность [13].

# Список литературы

[1] Воронин А. А., Смирнова Е. В., Смирнов А. П. К вопросу идентификации пород древесины с применением методов анализа спектров. // Научно-технический вестник СПБГУ ИТМО №2(66). 2010. 5–11 с.

[2] Зайдель А. Н. Основы спектрального анализа. // «Наука». 1965. 322 с.

[3] Haykin S. Neural Networks: A Comprehensive Foundation, 2nd edition. Hamilton, Ontario: Pearson Education. 1999. 823 p.

[4] Eigen, JacobiSVD.
http://eigen.tuxfamily.org/dox/classEigen_1_1JacobiSVD.html

[5] Логинов Н.В. Сингулярное разложение матриц. М. МГАПИ 1996. 80с.

[6] Jolliffe I. T. Principal Component Analysis. New York: Springer. 2002. 518 p.

[7] D. Wasserman. Neural Computing: Theory and Practice, First Edition. Coriolis Group (Sd); edition. 1989. 230 p.

[8] Айвазян С. А., Бухштабер В. М., Енюков И. С., Мешалкин Л. Д. Прикладная статистика: классификация и снижение размерности. М.: Финансы и статистика, 1989. 607 с.

[9] Bishop C. M. Pattern Recognition and Machine Learning. Springer, 2006. 738 p.

[10] B Stroustrup. The C++ Programming Language. 3rd edition. Addison–Wesley Pub. 2000, 1033 p.

[11] Бабушкин А.А. Методы спектрального анализа. МГУ, 1962. 509 с.

[12] T. Masters. Practical Neural Network Recipes in C++. Morgan Kaufmann; 1 edition. 1993. 493 p.

[13] К. В. Воронцов, Лекции по алгоритмическим композициям, 2012, 43 с. http://www.machinelearning.ru/wiki/images/0/0d/Voron-ML-Compositions.pdf

[14] Журавлев Ю. И., Рязанов В. В., Сенько О. В. «Распознавание». М.: Фазис, 2006. 176 с

[15] Выбор параметров многослойных нейронных сетей прямого распространения. http://mei06.narod.ru/sem7/iis/shpora/page2_9.htm

# Приложение

Net.cpp

```
#include "Net.h"

Net::~Net() {
}

Net::Net() {
}
//--------------------------------------------------------

//--------------------------------------------------------
// Public methods.!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
//--------------------------------------------------------

Net::Net(vector<int> conf) :config(conf) {
        for (int i = 1; i <= config[0]; i++) {
                Layer a(config[i], config[i + 1]);
                layers.push_back(a);
        }
};
//--------------------------------------------------------
Net::Net(vector<vector<vector<double>>> wieght)
{
        for (auto elemWieght : wieght) {
                Layer a(elemWieght);
                layers.push_back(a);
        }
}
//--------------------------------------------------------
Net::Net(string Path)
{
        ifstream in(Path);
        vector<int> conf;
        int read;
        in >> read;
        conf.push_back(read);
        for (int i = 0; i <= conf[0]; i++)
        {
                in >> read;
                conf.push_back(read);
        }
        config = conf;
        vector<vector<vector<double>>> wieght;
        vector<vector<double>> lay;
        vector<double> neu;

        double w;
        for (int i = 0; i < conf[0]; i++) {
                lay.clear();
                for (int j = 0; j < (conf[i + 2]); j++) {
                        neu.clear();
                        for (int k = 0; k <= (conf[i + 1]); k++)
                        {
                                in >> w;
```

```
                        neu.push_back(w);
                    }
                    lay.push_back(neu);
                }
                wieght.push_back(lay);
            }
            for (auto elementWieght : wieght) {
                Layer a(elementWieght);
                layers.push_back(a);
            }
}
//-------------------------------------------------------------
int Net::getNumberInputData() {
            return config[1];
}
//-------------------------------------------------------------
vector<vector<double>> Net::workResult(vector<vector<double>> x) {
            vector<vector<double>> buffReturn;
            for (auto xItem : x)
                buffReturn.push_back(startNet(xItem));
            return buffReturn;
}
//-------------------------------------------------------------
double Net::percentTrueAnswer(vector<vector<double>> xControl,
vector<vector<double>> dControl)
{
            double trueAnswer = 0;

            for (size_t i = 0; i < xControl.size(); i++) {
                vector<double> y = startNet(xControl[i]);
                int answerY = 0, answerD = 0;
                double maxY = 0, maxD = 0;
                for (decltype(y.size()) j = 0; j < y.size(); j++) {
                    if (y[j] > maxY) {
                        maxY = y[j];
                        answerY = j;
                    }
                    if (dControl[i][j] > maxD) {
                        maxD = dControl[i][j];
                        answerD = j;
                    }
                }
                if (answerD == answerY) {
                    trueAnswer++;
                }
            }
            return 100*(trueAnswer/ xControl.size());
}
//-------------------------------------------------------------
vector<double> Net::startNet(vector<double> x) {
            for (int i = 0; i < config[0]; i++) {
                x = layers[i].actF(x);
            }
            return x;
}
//-------------------------------------------------------------
int Net::teaching(vector<vector<double>> x, vector<vector<double>> d,
vector<vector<double>> testX, vector<vector<double>> testD, double e) {
            srand(time(0));
```

```
        std::cout << endl << "Start teach" << endl;
        int numberEpoch = 0;
        int     random_key = 0;
        int iter = 0;
        int errorControlEndNumber = 0;
        bool epochControl = true, diffControl = true, minControl = true,
standardDeviation = true;
        ofstream outErrouTeach(savePath + "\\error_teach.txt");
        double errorTeach = 10, errorMax = 0, errorMin = 5000, errorTest = 0,
errorMinTest = 100;
        vector<int> numberTeachItem(x.size(),0);
        list<double> diffTeachTestErrou, standardDeviationList;
        //---
        bool miniGold = false;
        //---
        // Begin teach.
        while (errorTeach > e && minControl && diffControl &&
standardDeviation) {
                for (auto v : numberTeachItem)
                        epochControl = epochControl && v;
                if (epochControl) {
                        iter = 0;
                        vector<int> a(x.size(), 0);
                        numberTeachItem = a;
                        numberEpoch++;
                        //save(numberEpoch);
                        errorTeach = 0;
                        errorTest = 0;
                        errorMax = 0;
                        errorMin = 5000;
                        // Calculating errors.
                        for (decltype(testX.size()) j = 0; j < testX.size();
j++)
                                errorTest = errorTest +
functionError(startNet(testX[j]), testD[j]);
                        errorTest = errorTest / testX.size();
                        if(errorMinTest > errorTest)
                                errorMinTest = errorTest;
                        for (decltype(x.size()) j = 0; j < x.size(); j++)
                        {
                                double buffErrou =
functionError(startNet(x[j]), d[j]);
                                errorTeach = errorTeach + buffErrou;
                                if (buffErrou > errorMax)
                                        errorMax = buffErrou;
                                if (buffErrou < errorMin)
                                        errorMin = buffErrou;
                        }
                        errorTeach = errorTeach / x.size();
                        // Error records by epoch number in console.
                        std::cout << endl << numberEpoch << " == " <<
errorTeach << endl;
                        std::cout << " TEST == " << errorTest << endl;
                        std::cout << " MinErrou = " << errorMin << " MaxErrou
= " << errorMax << endl;
                        // Error records by epoch number in file.
                        outErrouTeach << numberEpoch << " " << errorTeach <<
" " << errorTest << endl;
```

```
                        // Monitoring of the change in the minimum error on
the test set.
                        if (diffControl) {
                                if (numberEpoch == 1)
                                        for (size_t i = 0; i < 6; i++)

        diffTeachTestErrou.push_back(abs(errorTest - errorTeach));
                                else {
                                        diffTeachTestErrou.push_back(errorTest
- errorTeach);
                                        diffTeachTestErrou.pop_front();
                                }
                                double sumDiff = 0;
                                for (auto item : diffTeachTestErrou) {
                                        sumDiff += item;
                                }
                                if (sumDiff / 6 > 0.08)
                                        diffControl = false;
                        }
                        if (standardDeviation) {
                                if(numberEpoch == 1)
                                        for (size_t i = 0; i < 20; i++)

        standardDeviationList.push_back(errorTest);
                                else {

        standardDeviationList.push_back(errorTest);
                                        standardDeviationList.pop_front();
                                }
                                if (numberEpoch > 30) {
                                        double x_ =
accumulate(standardDeviationList.begin(), standardDeviationList.end(),
0)/20;
                                        double s = 0;
                                        for (auto stanDevItem:
standardDeviationList)
                                                s += (stanDevItem -
x_)*(stanDevItem - x_);
                                        if (sqrt(s / 20) < 0.003)
                                                standardDeviation = false;
                                }
                        }
                        if ((numberEpoch > 750) || ((numberEpoch > 700) &&
(errorMinTest + 0.001) >= errorTest)) {
                                minControl = false;
                                continue;
                        }
                }
                // New iteration.
                epochControl = true;
                iter++;
                do
                {
                        random_key = rand() % x.size();
                        numberTeachItem[random_key]++;
                } while (numberTeachItem[random_key]>3);

                std::cout << "\r\t\t\t\t\t\t\t\r=== Iter " << iter << "\t"<<
numberTeachItem[random_key];
```

```
                //-------
                if ((numberEpoch == 0) ||  (errorTeach - e < 0.05) ||
miniGold) {
                        goldEps = 0.01;
                        if ((errorTeach - e) < 0.05)
                                miniGold = true;
                }
                else if(numberEpoch < 200){
                        goldEps = 0.2 + (errorTeach - e) -
double(numberEpoch)/1000;
                }
                else{
                        goldEps = (errorTeach - e);
                }
                //------
                double buffErrou = teach(x[random_key], d[random_key]);
                std::cout << "\tENDteach ===";
                if (buffErrou < errorMin + (errorMax - errorMin)*0.70)
                        numberTeachItem[random_key] = 3;
        }
        outErrouTeach.close();
        save(0);
        return (0);
}
//-------------------------------------------------------------
void Net::setLayer(vector<Layer> lay) {
        layers = lay;
}
//-------------------------------------------------------------
double Net::functionError(vector<double> y, vector<double> d) {
        double errorf = 0;
        for (decltype(y.size()) i = 0; i < y.size(); i++) {
                errorf = errorf + (y[i] - d[i]) * (y[i] - d[i]);
        }
        return errorf / 2;
}

//-------------------------------------------------------------
// Private methods.!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
//-------------------------------------------------------------

vector<vector<double>> Net::deltaM(vector<double> d) {

        vector<double> deltalay;
        vector<vector<double>> delta(config[0]);
        vector<double> activf = layers[config[0] - 1].actF();
        vector<double> derivf = layers[config[0] - 1].derF();
        for (int i = 0; i < config[config.size() - 1]; i++) {
                deltalay.push_back((activf[i] - d[i])*derivf[i]);
                //deltalay.push_back((activf[i] - d[i])*(activf[i] *(1-
activf[i])));
        }
        delta[config[0] - 1] = deltalay;

        vector<vector<double>> matrix_w;
        for (int lay = config[0] - 2; lay >= 0; lay--) {
                deltalay.clear();
                activf.clear();
                matrix_w.clear();
```

```
                matrix_w = layers[lay + 1].getMatrixW();
                vector<double> activf = layers[lay].actF();
                vector<double> derivf = layers[lay].derF();
                double sum_del_w;
                for (int j = 0; j < config[lay + 2]; j++) {
                        sum_del_w = 0;
                        for (decltype(matrix_w.size()) k = 0; k <
matrix_w.size(); k++) {
                                sum_del_w = sum_del_w + delta[lay + 1][k] *
matrix_w[k][j];
                        }
                        deltalay.push_back(sum_del_w*derivf[j]);
                        //deltalay.push_back(sum_del_w*(activf[j] * (1 -
activf[j])));
                }
                delta[lay] = deltalay;
        }
        return delta;
}
//-------------------------------------------------------------
void Net::correct(vector<vector<double>> delta, double alfa, vector<double>
x) {
        layers[0].correct(delta[0], x, alfa);
        for (int i = 1; i < config[0]; i++) {
                layers[i].correct(delta[i], layers[i - 1].actF(), alfa);
        }
}
//-------------------------------------------------------------
double Net::goldenSection(vector<vector<double>> delta, vector<double> x,
vector<double> d) {
        Net minimiNet(config);
        minimiNet.setLayer(layers);
        double fi = (1 + sqrt(5)) / 2;
        double a = 0;
        double b = 1;
        double x1, x2, f1, f2;
        vector<double> y;
        x1 = b - (b - a) / fi;
        x2 = a + (b - a) / fi;
        while (abs(b - a) > goldEps) {

                minimiNet.setLayer(layers);//  !
                minimiNet.correct(delta, x1, x);
                y = minimiNet.startNet(x);
                f1 = minimiNet.functionError(y, d);

                minimiNet.setLayer(layers);//  !
                minimiNet.correct(delta, x2, x);
                y = minimiNet.startNet(x);
                f2 = minimiNet.functionError(y, d);

                if (f1 > f2) {
                        a = x1;
                        x1 = x2;
                        x2 = a + (b - a) / fi;
                }
                else {
                        b = x2;
                        x2 = x1;
```

```
                        x1 = b - (b - a) / fi;
                }
        }
        return (a + b) / 2;
}
//-------------------------------------------------------------
double Net::teach(vector<double> x, vector<double> d) {
        vector<double> y = startNet(x);
        vector<vector<double>> delta = deltaM(d);
        double alfa = goldenSection(delta, x, d);
        correct(delta, alfa, x);
        return (functionError(y, d));
}
//-------------------------------------------------------------
vector<vector<vector<double>>> Net::save(int i) {
        vector<vector<vector<double>>> W;
        ofstream outs(savePath+"\\saveWeight" + to_string(i) + ".txt");
        for (int i = 0; i < config[0]; i++)
        {
                vector<vector<double>> a = layers[i].getMatrixW();
                W.push_back(a);
        }
        for (decltype(config.size()) i = 0; i < config.size(); i++) {
                outs << config[i] << " ";
        }
        outs << endl;
        for (decltype(W.size()) i = 0; i < W.size(); i++) {
                for (decltype(W[0].size()) j = 0; j < W[i].size(); j++) {
                        for (decltype(W[0][0].size()) k = 0; k <
W[i][j].size(); k++)
                                outs << W[i][j][k] << " ";
                        outs << endl;
                }
        }
        outs.close();
        return(W);
}
```

Layer.cpp

```
#include "Layer.h"

Layer::~Layer() {
}
Layer::Layer() {
}

Layer::Layer(int early_kol_neu, int kol_neu) {
        for (int i = 0; i < kol_neu; i++) {
                Neuron a(early_kol_neu);
                neurons.push_back(a);
                activ_f.push_back(0);
                sum_s.push_back(0);
        }
}
```

```
Layer::Layer(vector<vector<double>> W) {
	for (decltype(W.size()) i = 0; i < W.size(); i++) {
		Neuron a(W[i]);
		neurons.push_back(a);
		activ_f.push_back(0);
		sum_s.push_back(0);
	}
}

vector<double> Layer::actF(vector<double> X) {
	for (decltype(neurons.size()) i = 0; i < neurons.size(); i++) {
		activ_f[i] = neurons[i].ActF(X);
		sum_s[i] = neurons[i].Sum();
	}
	return(activ_f);
}

vector<double> Layer::actF() {
	return(activ_f);
}

vector<double> Layer::derF() {
	vector<double> getBack;
	for (auto neuron : neurons)
		getBack.push_back(neuron.derF());
	return(getBack);
}


int Layer::correct(vector<double> deltw, vector<double> y, double alfa) {
	for (decltype(neurons.size()) i = 0, sz = neurons.size(); i < sz;
i++) {
		neurons[i].correctWeights(deltw[i], y, alfa);
	}
	return 0;
}

vector<double> Layer::getVectorW(int i) { //
	return neurons[i].getAllWeights();
}

vector<vector<double>> Layer::getMatrixW() { //
	vector<vector<double>> matrix;
	for (decltype(neurons.size()) i = 0; i<neurons.size(); i++) {
		matrix.push_back(getVectorW(i));
	}
	return matrix;
}
```

Neuron.cpp

```
#include "Neuron.h"

Neuron::Neuron(int number_weight) {
	srand(time(0));
	for (int i = 0; i <= number_weight; i++) //
		weights.push_back(0.0001 * (rand() % 6001 - 3000));  //
```

```
        sum = 0;
        F = 0;
}
//-------------------------------------------------------
Neuron::Neuron(vector<double> verton_weight) {
        weights = verton_weight;
        sum = 0;
        F = 0;
}
//-------------------------------------------------------
Neuron::~Neuron() {
}

Neuron::Neuron() {
}
//-------------------------------------------------------
vector<double> Neuron::getAllWeights() { //
        return weights;
}
//-------------------------------------------------------
double Neuron::getElemWeight(int number) {
        return(weights[number]);
}
//-------------------------------------------------------
double Neuron::Sum(vector<double> x) {
        sum = weights[weights.size() - 1];
        for (decltype(weights.size()) i = 0; i < weights.size() - 1; i++)
                sum = sum + weights[i] * x[i];
        return sum;
}
//-------------------------------------------------------
double Neuron::Sum() {
        return sum;
}
//-------------------------------------------------------

/*
double Neuron::ActF(vector<double> x) {
        Sum(x);
        F = (exp(sum) - exp(-sum)) / (exp(sum) + exp(-sum));
        return F;
}
*/
double Neuron::ActF(vector<double> x) {
        Sum(x);
        F = 1 / (1 + exp(-sum));
        return F;
}
//-------------------------------------------------------
double Neuron::ActF() {
        return F;
}
//-------------------------------------------------------
double Neuron::derF() {
        return (1 - F)*F;
        //return 1-F*F;
}
//-------------------------------------------------------
void Neuron::correctWeights(double deltaW, vector<double> y, double alfa) {
```

```
	for (decltype(weights.size()) i = 0; i < weights.size() - 1; i++)
		weights[i] = weights[i] - deltaW * y[i] * alfa;
	weights[weights.size() - 1] = weights[weights.size() - 1] - deltaW *
alfa;
}
//-------------------------------------------------------------
void Neuron::setNewVectorWeights(vector<double> W) {
	weights = W;
}
```

DataSet.cpp

```
#include "DataSet.h"

//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++//
ParseData::ParseData() {

}
//-------------------------------------------------------------
ParseData::~ParseData() {

}
//-------------------------------------------------------------
Bloc::Bloc() {

}
//-------------------------------------------------------------
Bloc::~Bloc() {

}
//-------------------------------------------------------------
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++//
double divider(double a) {
	if (abs(a) > 10)
		return(100);
	if (abs(a) > 0.88)
		return(10);
	return(1);
}

void divisionComponents(vector<vector<double>> &x, double buffDiv1) {

	for (size_t i = 0; i < x.size(); i++)
		x[i][0] = x[i][0] / buffDiv1;
}
//-------------------------------------------------------------
// Public methods.!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
//-------------------------------------------------------------

/*
	Auxiliary function.
	Returns everything in the folder (names).
*/
vector<string> ParseData::getDirectoryAttachments(string dPath) {
	path p(dPath);
```

```
        vector<string> v;
        for (auto&& x : directory_iterator(p))
                v.push_back(x.path().filename().string());
        sort(v.begin(), v.end());
        return v;
}
//-------------------------------------------------------
vector<Data> ParseData::getDataTEST() {
        vector<Data> getBack;
        Data teach, test, control;
        size_t sizeblocSet = 0, sizeSum = 0;

        for (auto sizeData : numberDataItems) {
                // Distribution of data between sets.

                size_t  sizeTeach, sizeTest, sizeControl;
                sizeTeach = sizeData*0.6;
                sizeTest = (sizeData - sizeTeach)*0.5;
                sizeControl = sizeData - sizeTeach - sizeTest;
                for (/**/; sizeblocSet < sizeSum + sizeData; sizeblocSet++) {
                        if (sizeblocSet < sizeSum+ sizeTeach) {
                                for (auto v : blocSet[sizeblocSet].data) {
                                        teach.data.push_back(v);

        teach.answer.push_back(dataAnswer[sizeblocSet]);
                                }
                        }
                        else if(sizeblocSet < sizeSum + sizeTeach+ sizeTest){
                                for (auto v : blocSet[sizeblocSet].data) {
                                        test.data.push_back(v);

        test.answer.push_back(dataAnswer[sizeblocSet]);
                                }
                        }
                        else {
                                for (auto v : blocSet[sizeblocSet].data) {
                                        control.data.push_back(v);

        control.answer.push_back(dataAnswer[sizeblocSet]);
                                }
                        }
                }
                sizeSum += sizeData;
        }
        getBack.push_back(teach);
        getBack.push_back(test);
        getBack.push_back(control);
        return getBack;
}
//-------------------------------------------------------------
vector<Data> ParseData::getDataCrossValid(vector<CrossValid> crossV) {
        Data teach, test, control;
        size_t sizeSum = 0;
        for (size_t numberClass = 0; numberClass < crossV.size();
numberClass++) {

                for (auto i: crossV[numberClass].teach)
                        for (auto v : blocSet[sizeSum + i].data) {
                                teach.data.push_back(v);
```

```
                                teach.answer.push_back(dataAnswer[sizeSum +
i]);
                        }
                for (auto i : crossV[numberClass].test)
                        for (auto v : blocSet[sizeSum + i].data) {
                                test.data.push_back(v);
                                test.answer.push_back(dataAnswer[sizeSum +
i]);
                        }
                for (auto i : crossV[numberClass].control)
                        for (auto v : blocSet[sizeSum + i].data) {
                                control.data.push_back(v);
                                control.answer.push_back(dataAnswer[sizeSum


+ i]);
                        }
                sizeSum += numberDataItems[numberClass];
        }
        vector<Data> buffReturn;
        buffReturn.push_back(teach);
        buffReturn.push_back(test);
        buffReturn.push_back(control);
        return buffReturn;
}
//-------------------------------------------------------------
vector<int> ParseData::getClassDistribution() {
        return(numberDataItems);
}
//-------------------------------------------------------------
Bloc::Bloc(string fPath) {

        path p(fPath);
        // If read from the file at once.
        if (exists(p)) {
                if (is_regular_file(p)) {
                        std::ifstream in(fPath);
                        cout << "        " + fPath << endl;
                        string bufData;
                        while (getline(in, bufData))
                        {
                                vector<double> dataElem;
                                size_t posR, posL;
                                posL = 0;
                                while (posL != bufData.find_last_of(" ")) {
                                        posR = bufData.find(" ", posL + 1);
                                        if (posL > 0)
                                                dataElem.push_back(stod(
bufData.substr(posL + 1, posR - posL - 1)));
                                        else

        dataElem.push_back(stod(bufData.substr(posL,
 posR - posL)));
                                        posL = posR;
                                }
                                data.push_back(dataElem);
                                dataElem.clear();
                        }
```

```
                }
        }
        // If there are several files.
        else {
                int numberOfScan = 1;
// Counting the number of files for one instance, the path to the file is
built on it.
                path p(fPath + " 0" + to_string(numberOfScan));

                //      Passage on files while there are supposed files.
                while (exists(p)) {
                        vector<double> dataElem;
                        std::ifstream in;
                        if (numberOfScan < 10) {
                                in.open(fPath + " 0" +
to_string(numberOfScan));
                                cout << "Read file: " + fPath + " 0"
 + to_string(numberOfScan) << endl;
                        }
                        else {
                                in.open(fPath + " " +
to_string(numberOfScan));
                                cout << "Read file: " + fPath + " "
+ to_string(numberOfScan) << endl;
                        }
                        string Elem;

                        // Reading a file.
                        while (getline(in, Elem)) {

        dataElem.push_back(stod(Elem.substr((Elem.find(" ") + 1))));
                        }

                        data.push_back(dataElem);

                        ++numberOfScan;
                        if (numberOfScan < 10)
                                p = path(fPath + " 0" +
to_string(numberOfScan));
                        else
                                p = path(fPath + " " +
to_string(numberOfScan));
                        dataElem.clear();
                        in.close();
                }
        }
}
//-------------------------------------------------------------
ParseData::ParseData(string dPath) {
        path directWithClasses(dPath);

        if (is_directory(directWithClasses))
        {
                cout << directWithClasses << " is a directory containing
Classes :" << endl;
                vector<string> classesData = getDirectoryAttachments(dPath);

                size_t numberC = 0;
//      Number of the current class to form a dictionary with answers
```

```
                for (auto&& className : classesData) {

                        // Forming a dict with answers.
                        vector<double> d(classesData.size(), 0.05);
                        d[numberC] = 0.95;
                        ++numberC;
                        mapAnswer.insert(pair<vector<double>, string>(d,
className));

                        //      Path to a folder with files for a particular
class.
                        string classPath = dPath + "\\" + className;
                        //
                        vector<string> classElem =
getDirectoryAttachments(classPath);
                        int kolElem = stoi(classElem[classElem.size() -
1].substr(0, 2));
                        numberDataItems.push_back(kolElem);
                        //numberClassItems.push_back(kolElem);
                        for (int i = 1; i <= kolElem; i++) {
                                if (i < 10) {
                                        Bloc a(classPath + "\\0" +
to_string(i));
                                        blocSet.push_back(a);
                                        dataAnswer.push_back(d);
                                }
                                else {
                                        Bloc a(classPath + "\\" +
to_string(i));
                                        blocSet.push_back(a);
                                        dataAnswer.push_back(d);
                                }
                        }
                        d.clear();
                }

                cout << endl << "      Create DATA END" << endl;
        }
        else {
                cout << endl << "      Path: " + dPath + " is a not
directory" << endl;
        }
}
//-------------------------------------------------------------
//Cross validatiom methots.
//-------------------------------------------------------------
vector<CrossValid> CrossValidation(vector<int> classDistribution) {
        //srand(time(0));
        vector<CrossValid> buffReturn;
        for (int sizeData : classDistribution) {
                CrossValid buffPush;
                vector<int> vForRandCreate;
                for (int i = 0; i < sizeData; i++)
                        vForRandCreate.push_back(i);
                size_t  sizeTeach, sizeTest, sizeControl;
                sizeTeach = sizeData*0.6;
                sizeTest = (sizeData - sizeTeach)*0.5;
                sizeControl = sizeData - sizeTeach - sizeTest;
                //
```

```
                while (vForRandCreate.size() > 0) {
                        //srand(time(0));
                        int randomKey = 0;
                        if (vForRandCreate.size() > 1)
                                randomKey = rand() % (vForRandCreate.size() -
1);

                        if (vForRandCreate.size() > (sizeTest + sizeControl))
{

        buffPush.teach.push_back(vForRandCreate[randomKey]);
                                vForRandCreate.erase(vForRandCreate.begin() +
randomKey);
                        }
                        else if (vForRandCreate.size() > sizeControl) {

        buffPush.test.push_back(vForRandCreate[randomKey]);
                                vForRandCreate.erase(vForRandCreate.begin() +
randomKey);
                        }
                        else {

        buffPush.control.push_back(vForRandCreate[randomKey]);
                                vForRandCreate.erase(vForRandCreate.begin() +
randomKey);

                        }
                }
                buffReturn.push_back(buffPush);
        }
        return buffReturn;
}
//-------------------------------------------------------------
void saveCrossValid(string Path, vector<CrossValid> v) {
        std::ofstream out(Path);
        out << v.size() << endl;
        for (size_t i = 0; i < v.size(); i++) {
                out << v[i].teach.size() << " " << v[i].test.size() << " " <<
v[i].control.size() << endl;
                std::copy(v[i].teach.begin(), v[i].teach.end(),
 std::ostream_iterator<int>(out, " "));
                out << endl;
                std::copy(v[i].test.begin(), v[i].test.end(),
 std::ostream_iterator<int>(out, " "));
                out << endl;
                std::copy(v[i].control.begin(), v[i].control.end(),
 std::ostream_iterator<int>(out, " "));
                out << endl;
        }
}
//-------------------------------------------------------------
vector<CrossValid> readCrossValid(string Path) {
        std::ifstream in(Path);
        vector<CrossValid> buffReturn;
        int kol;
        in >> kol;
        for (int i = 0; i < kol; i++) {
                CrossValid buffPush;
                int kolTeach, kolTest, kolControl;
```

```
                in >> kolTeach >> kolTest >> kolControl;
                for (int j = 0; j < kolTeach; j++) {
                        int b;
                        in >> b;
                        buffPush.teach.push_back(b);
                }
                for (int j = 0; j < kolTest; j++) {
                        int b;
                        in >> b;
                        buffPush.test.push_back(b);
                }
                for (int j = 0; j < kolControl; j++) {
                        int b;
                        in >> b;
                        buffPush.control.push_back(b);
                }
                buffReturn.push_back(buffPush);
        }
        return buffReturn;
}
```

---

moduleSVD.cpp

---

```
#include "moduleSVD.h"

moduleSVD::moduleSVD()
{
}

moduleSVD::~moduleSVD()
{
}

//-----------------------------------------------------------
// Public methods.!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
//-----------------------------------------------------------

//-----------------------------------------------------------
moduleSVD::moduleSVD(vector<vector<double>> teachData) {
        teachMatrix = vectVectToMatrix(teachData);
        JacobiSVD<MatrixXd> svd(teachMatrix, ComputeThinU | ComputeThinV);
        MatrixXd singular(svd.matrixU().cols(), svd.matrixU().cols());
        for (int i = 0; i < svd.matrixU().cols(); i++)
                for (int j = 0; j < svd.matrixU().cols(); j++)
                        if (i == j)
                                singular(i, j) = svd.singularValues()(i);
                        else
                                singular(i, j) = 0;
        T = svd.matrixU() * singular;
        P = svd.matrixV();
}

//-----------------------------------------------------------
vector<vector<double>> moduleSVD::getTeachData(int components) {
        return matxrixToVectVect(T.block(0,0,T.rows(),components));
```

```
}
//-------------------------------------------------------

vector<vector<double>> moduleSVD::getNewhData(int components,
vector<vector<double>> data) {

	return
matxrixToVectVect((vectVectToMatrix(data)*P).block(0,0,data.size(),components
));
}

//-------------------------------------------------------
// Private methods.!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
//-------------------------------------------------------

//-------------------------------------------------------
vector<vector<double>> moduleSVD::matxrixToVectVect(MatrixXd m) {
	vector<vector<double>> getBack;
	vector<double> buf;
	for (size_t i = 0; i < m.rows(); i++) {
		for (size_t j = 0; j < m.cols(); j++) {
			buf.push_back(m(i, j));
		}
		getBack.push_back(buf);
		buf.clear();
	}
	return (getBack);
}

//-------------------------------------------------------
MatrixXd moduleSVD::vectVectToMatrix(vector<vector<double>> vV) {
	MatrixXd m(vV.size(), vV[0].size());
	for (size_t j = 0; j < vV[0].size(); j++)
		for (size_t i = 0; i < vV.size(); i++)
			m(i, j) = vV[i][j];
	return m;
}
```

SpectralClassifier.cpp

```
#include "SpectralClassifier.h"



SpectralClassifier::SpectralClassifier()
{
}

SpectralClassifier::~SpectralClassifier()
{
}
//-------------------------------------------------------

//-------------------------------------------------------
SpectralClassifier::SpectralClassifier(bool teachOrRead,string Path, string
dataPath):PathS(Path){
	// Инициализация сетей.
```

```
        if(!teachOrRead){
                {Net newNet(Path + "\\ir\\saveWeight0.txt");
                irNet = newNet; }
                {Net newNet(Path + "\\diod\\saveWeight0.txt");
                diodNet = newNet; }
                {Net newNet(Path + "\\visible\\saveWeight0.txt");
                visibleNet = newNet; }
        }
        else {
                int a, b;/*
                cout << "\nNumber input data & number neuron:\n";
                cin >> a >> b;
                vector<int> settings_net = { 2,a,b,14 };
                */
                {cout << "\nNumber input data & number neuron for ir Net:\n";
                cin >> a >> b;
                vector<int> settings_net = { 2,a,b,14 };
                Net newNet(settings_net);
                irNet = newNet;
                irNet.savePath = Path + "\\ir";}
                //------------------
                {cout << "\nNumber input data & number neuron for diod
Net:\n";
                cin >> a >> b;
                vector<int> settings_net = { 2,a,b,14 };
                Net newNet(settings_net);
                diodNet = newNet;
                diodNet.savePath = Path + "\\diod"; }

                {cout << "\nNumber input data & number neuron for visible
net:\n";
                cin >> a >> b;
                vector<int> settings_net = { 2,a,b,14 };
                Net newNet(settings_net);
                visibleNet = newNet;
                visibleNet.savePath = Path + "\\visible"; }
        }
        // Создание главной сети.
        {vector<int> conf = { 1,42,14 };
        Net newNet(conf);
        classifier = newNet;
        classifier.savePath = Path + "\\Classifier";
        }

        // Обработка данных.
        ParseData dataParsIr(dataPath + "\\ir");
        ParseData dataParsDiod(dataPath + "\\diod");
        ParseData dataParsVisible(dataPath + "\\visible");

        vector<CrossValid> generalCross =
readCrossValid(Path+"\\crossValid.txt");
        vector<Data> dataIr = dataParsIr.getDataCrossValid(generalCross);
        vector<Data> dataDiod = dataParsDiod.getDataCrossValid(generalCross);
        vector<Data> dataVisible =
dataParsVisible.getDataCrossValid(generalCross);
        cout << "start SVD" << endl;
        moduleSVD svdIr(dataIr[0].data);
        moduleSVD svdDiod(dataDiod[0].data);
        moduleSVD svdVisible(dataVisible[0].data);
```

```
        cout << "end SVD" << endl;
        Data buffPush;

        //-------------------------------------------------------------
        if (teachOrRead) {
                teachOneClassifier(visibleNet, svdVisible, dataVisible, 0.06);
                teachOneClassifier(irNet, svdIr, dataIr, 0.07);
                teachOneClassifier(diodNet, svdDiod, dataDiod, 0.195);
        }
        //-------------------------------------------------------------
        vector<vector<double>> xIr,xDiod,xVisible;
        // Ir1

        xIr = svdIr.getTeachData(irNet.getNumberInputData());
        double xIrDiv = divider(xIr[0][0]);
        divisionComponents(xIr, xIrDiv);
        buffPush.data = irNet.workResult(xIr);

        // Diod1
        xDiod = svdDiod.getTeachData(diodNet.getNumberInputData());
        double xDiodDiv = divider(xDiod[0][0]);
        divisionComponents(xDiod, xDiodDiv);
        int i = 0;
        for (auto x : diodNet.workResult(xDiod)) {
                for(auto xItem: x)
                        buffPush.data[i].push_back(xItem);
                i++;
        }

        // Visible1
        i = 0;
        xVisible = svdVisible.getTeachData(visibleNet.getNumberInputData());
        double xVisibleDiv = divider(xVisible[0][0]);
        divisionComponents(xVisible,xVisibleDiv);
        for (auto x : visibleNet.workResult(xVisible)) {
                for (auto xItem : x)
                        buffPush.data[i].push_back(xItem);
                i++;
        }
        buffPush.answer = dataVisible[0].answer;
        dataCollegium.push_back(buffPush);
        cout << "clr" << endl;
        buffPush.clear();

        //-------------------------------------------------------------------
-------
        // Ir2
        xIr = svdIr.getNewhData(irNet.getNumberInputData(), dataIr[1].data);
        divisionComponents(xIr, xIrDiv);
        buffPush.data = irNet.workResult(xIr);

        // Diod2
        xDiod = svdDiod.getNewhData(diodNet.getNumberInputData(),
dataDiod[1].data);
        divisionComponents(xDiod, xDiodDiv);
        i = 0;
        for (auto x : diodNet.workResult(xDiod)) {
                for (auto xItem : x)
                        buffPush.data[i].push_back(xItem);
```

```
                i++;
        }

        // Visible2
        xVisible = svdVisible.getNewhData(visibleNet.getNumberInputData(),
dataVisible[1].data);
        divisionComponents(xVisible, xVisibleDiv);
        i = 0;
        for (auto x : visibleNet.workResult(xVisible)) {
                for (auto xItem : x)
                        buffPush.data[i].push_back(xItem);
                i++;
        }
        buffPush.answer = dataVisible[1].answer;
        dataCollegium.push_back(buffPush);
        buffPush.clear();
        //-------------------------------------------------------------------
---
        // Ir3
        std::ofstream out(Path + "\\percentTrue_3_Net.txt");

        xIr = svdIr.getNewhData(irNet.getNumberInputData(), dataIr[2].data);
        divisionComponents(xIr, xIrDiv);
        buffPush.data = irNet.workResult(xIr);
        out << "Ir: " << irNet.percentTrueAnswer(xIr, dataVisible[2].answer)
<< endl;

        // Diod3
        xDiod = svdDiod.getNewhData(diodNet.getNumberInputData(),
dataDiod[2].data);
        divisionComponents(xDiod, xDiodDiv);
        i = 0;
        for (auto x : diodNet.workResult(xDiod)) {
                for (auto xItem : x)
                        buffPush.data[i].push_back(xItem);
                i++;
        }
        out << "Diod: " << diodNet.percentTrueAnswer(xDiod,
dataVisible[2].answer) << endl;

        // Visible3
        i = 0;
        xVisible = svdVisible.getNewhData(visibleNet.getNumberInputData(),
dataVisible[2].data);
        divisionComponents(xVisible, xVisibleDiv);
        for (auto x : visibleNet.workResult(xVisible)) {
                for (auto xItem : x)
                        buffPush.data[i].push_back(xItem);
                i++;
        }
        buffPush.answer = dataVisible[2].answer;
        out << "Visible: " << visibleNet.percentTrueAnswer(xVisible,
dataVisible[2].answer) << endl;
        dataCollegium.push_back(buffPush);
        //---------------------
        buffPush.clear();
        //---------------------
}
//-------------------------------------------------------------
```

```
void SpectralClassifier::teachSR(double e) {
	classifier.teaching(dataCollegium[0].data, dataCollegium[0].answer,
dataCollegium[1].data, dataCollegium[1].answer, e);
	std::ofstream out(PathS + "\\percentTrueCR.txt");
	out <<
classifier.percentTrueAnswer(dataCollegium[2].data,dataCollegium[2].answer);
}
//-------------------------------------------------------------
void SpectralClassifier::teachSR_multi(int neuron,double e) {
	vector<int> conf = { 2,42,neuron,14 };
	Net newNet(conf);
	classifier = newNet;
	classifier.savePath = PathS + "\\" + to_string(neuron);
	classifier.teaching(dataCollegium[0].data, dataCollegium[0].answer,
dataCollegium[1].data, dataCollegium[1].answer, e);
	std::ofstream out(PathS + "\\" + to_string(neuron) +
"\\percentTrueCR.txt");
	out << classifier.percentTrueAnswer(dataCollegium[2].data,
dataCollegium[2].answer);
}

//-------------------------------------------------------------
void SpectralClassifier::teachOneClassifier(Net &generalNet, moduleSVD &svd,
vector<Data> &data, double e) {
	vector<vector<double>> x, testX, d, testD;

	x = svd.getTeachData(generalNet.getNumberInputData());
	d = data[0].answer;
	testX = svd.getNewhData(generalNet.getNumberInputData(),
data[1].data);
	testD = data[1].answer;

	divisionComponents(testX, divider(x[0][0]));
	divisionComponents(x, divider(x[0][0]));

	generalNet.teaching(x, d, testX, testD, e);
}
```